# *Chapter TWENTY-EIGHT*
# Fork/Join Framework

### *Exam Objectives*

*Use parallel Fork/Join Framework.*

## The Fork/Join Framework

The Fork/Join framework is designed to work with large tasks that can be split up into smaller tasks.

This is done through recursion, where you keep splitting up the task until you meet the base case, a task so simple that can be solved directly, and then combining all the partial results to compute the final result.

Splitting up the problem is know as **FORKING** and combining the results is known as **JOINING**.

The main class of the Fork/Join framework is `java.util.concurrent.ForkJoinPool`, which is actually a subclass of `ExecutorService`.

We create a `ForkJoinPool` instance mostly through two constructors:

```
ForkJoinPool()
ForkJoinPool(int parallelism)
```

The first version is the recommended way. It creates an instance with a number of threads equal to the number of processors of your machine (using the method `Runtime.getRuntime().availableProcessors()`).

The other version allows you to define the number of threads that will be used.

Just like an `ExecutorService` executes a task represented by a `Runnable` or a `Callable`, in the Fork/Join framework a task is usually represented by a subclass of either:

- `RecursiveAction`, which is the equivalent of `Runnable` in the sense that it **DOESN'T** return a value.
- `RecursiveTask<V>`, which is the equivalent of `Callable` in the sense that it **DOES** return a value.

Both of them extend from the abstract class `java.util.concurrent.ForkJoinTask`.

However, unlike the worker threads that an `ExecutorService` uses, the threads of a `ForkJoinPool` use a work-stealing algorithm, which means that when a thread is free, it **STEALS** the pending work of other threads that are still busy doing other work.

To implement this, three methods of your `ForkJoinTask` -based class are important to the framework:

```
ForkJoinTask<V> fork()
V join()
// if you extend RecursiveAction
protected abstract void compute()
// if you extend RecursiveTask
protected abstract V compute()
```

And each thread in the `ForkJoinPool` has a queue of these tasks.

In the beginning, you have a large task. This task is divided into (generally) two smaller tasks recursively until the base case is reached.

Each time a task is divided, you call the `fork()` method to place the first subtask in the current thread's queue, and then you call the `compute()` method on the second subtask (to recursively calculate the result).

This way, the first subtask will wait in the queue to be processed or stolen by an idle thread to repeat the process. The second subtask will be processed immediately (also repeating the process).

Of course, you have to divide the task enough times to keep all threads busy (preferably into a number of tasks greater that the number of threads to ensure this).

All right, let's review this. The first subtask is waiting in a queue to be processed, and the second one is processed immediately. So when or how do you get the result of the first subtask?

To get the result of the first subtask you call the `join()` method on this first subtask.

This should be the last step because `join()` will block the program until the result is returned.

This means that the **ORDER** in which you call the methods is **IMPORTANT**.

If you don't call `fork()` before `join()`, there won't be any result to get.

If you call `join()` before `compute()`, the program will perform like if it was executed in one thread and you'll be wasting time, because while the second subtask is recursively calculating the value, the first one can be stolen by another thread to process it. This way, when `join()` is finally called, either the result is ready or you don't have to wait a long time to get it.

But remember, the Fork/Join framework is not for every task. You can use it for any task that can be solved (or algorithm that can be implemented) recursively, but it's best for tasks that can be divided into smaller subtasks **AND** that they can be computed independently (so the order doesn't matter).

So let's pick a simple example, finding the minimum value of an array. This array can be split up in many subarrays and locate the minimum of each of them. Then, we can find the minimum value between those values.

Let's code this example with a `RecursiveAction` first, to see how this fork/join works. Remember that this class doesn't return a result so we're only going to print the partial results.

Another thing. The most basic scenario we can have (the base case) is when we just have to compare two values. However, having subtasks that are too small won't perform well.

For that reason, when working with fork/join, generally you divide the elements in sets of a certain size (that can be handled by a single thread), for which you solve the problem sequentially.

For this example, let's process five numbers per thread:

```
class FindMinimumAction extends RecursiveAction {
   // A thread can easily handle, let's say, five elements
   private static final int SEQUENTIAL_THRESHOLD = 5;
   // The array with the numbers (we'll pass the same array in
   // every recursive call to avoid creating a lot of arrays)
   private int[] data;
   // The index that tells use where a (sub)task starts
   private int start;
   // The index that tells use where a (sub)task ends
   private int end;

   // Since compute() doesn't take parameters, you have to
   // pass in the task's constructor the data to work
   public FindMinimumAction(int[] data, int start, int end) {
      this.data = data;
      this.start = start;
      this.end = end;
   }
   @Override
   protected void compute() {
      int length = end - start;
      if (length <= SEQUENTIAL_THRESHOLD) { // base case
         int min = computeMinimumDirectly();
         System.out.println("Minimum of this subarray: "+ min);
      } else { // recursive case
         // Calcuate new subtasks range
         int mid = start + length / 2;
         FindMinimumAction firstSubtask =
            new FindMinimumAction(data, start, mid);
         FindMinimumAction secondSubtask =
            new FindMinimumAction(data, mid, end);
         firstSubtask.fork(); // queue the first task
         secondSubtask.compute(); // compute the second task
         firstSubtask.join(); // wait for the first task result
      }
   }
   /** Method that find the minimum value */
   private int computeMinimumDirectly() {
      int min = Integer.MAX_VALUE;
      for (int i = start; i < end; i++) {
         if (data[i] < min) {
            min = data[i];
         }
      }
      return min;
   }
}
```

The `compute()` method defines the base case, when the (sub)array has five elements or less, in which case the minimum is found sequentially. Otherwise, the array is split into two subarrays recursively until the condition of the base case is fulfilled.

Dividing the tasks may not always result in evenly distributed subtasks. But to keep this simple, let's try the class with twenty elements, which is very likely to be split up into four sets:

```
public static void main(String[] args) {
   int[] data = new int[20];
   Random random = new Random();
   for (int i = 0; i < data.length; i++) {
      data[i] = random.nextInt(1000);
      System.out.print(data[i] + " ");
      // Let's print each subarray in a line
      if( (i+1) % SEQUENTIAL_THRESHOLD == 0 ) {
         System.out.println();
      }
   }
   ForkJoinPool pool = new ForkJoinPool();
   FindMinimumAction task =
      new FindMinimumAction(data, 0, data.length);
   pool.invoke(task);
}
```

A possible output can be:

```
109 411 348 938 776
188 42 28 818 825
642 454 431 742 463
33 832 705 910 456
Minimum of this subarray: 33
Minimum of this subarray: 28
Minimum of this subarray: 431
Minimum of this subarray: 109
```

Notice that we didn't need to shut down the pool explicitly. When the program exits, the pool is shut down, so it can be reused.

We also have the `invokeAll()` method, that doesn't return a value but does something equivalent to the call of `fork()`, `compute()`, and `join()` methods. So instead of having something like:

```
...
FindMinimumAction firstSubtask =
   new FindMinimumAction(data, start, mid);
FindMinimumAction secondSubtask =
   new FindMinimumAction(data, mid, end);
// queue the first task
firstSubtask.fork();
// compute the second task
secondSubtask.compute();
// wait for the first task result
firstSubtask.join();
...
```

We can simply have:

```
...
FindMinimumAction firstSubtask =
   new FindMinimumAction(data, start, mid);
FindMinimumAction secondSubtask =
   new FindMinimumAction(data, mid, end);
invokeAll(firstSubtask, secondSubtask);
...
```

Now, let's change this example to use a `RecursiveTask` so we can return the minimum value of all.

Actually, the only changes we need to do are in the `compute()` method:

```
class FindMinimumTask extends RecursiveTask<Integer> {
   // ...
   @Override
   //Return type matches the generic
   protected Integer compute() {
      int length = end - start;
      if (length <= SEQUENTIAL_THRESHOLD) { // base case
         return min = computeMinimumDirectly();
      } else { // recursive case
         // Calcuate new subtasks range
         int mid = start + length / 2;
         FindMinimumAction firstSubtask =
            new FindMinimumAction(data, start, mid);
         FindMinimumAction secondSubtask =
            new FindMinimumAction(data, mid, end);
         // queue the first task
         firstSubtask.fork();
         // Return the miminum of all subtasks
         return Math.min(firstSubtask.compute(),
                         secondSubtask.join());
      }
   }
   // ...
}
```

In the `main()` method, the only changes are printing the value that `pool.invoke(task)` returns and a little formatting to the output, which can be:

Array values:

```
819 997 124 425 669 657 487 447 386 979 31 748 194 644 893 209 913 810 142 565
Minimum value: 31
```

# Key Points

- The Fork/Join framework is designed to work with large tasks that can be split up into smaller tasks.
- This is done through recursion, where you keep splitting up the task until you meet the base case, a task so simple that can be solved directly, and then combining all the partial results to compute the final result.
- Splitting up the problem is know as **FORKING** and combining the results is known as **JOINING**.
- The main class of the Fork/Join framework is `java.util.concurrent.ForkJoinPool`, which is actually a subclass of `ExecutorService`.
- Just like an `ExecutorService` executes a task represented by a `Runnable` or a `Callable`, in the Fork/Join framework a task is represented by a subclass of either `RecursiveAction` (that **DOESN'T** return a value) or `RecursiveTask<V>` (that **DOES** return a value).
- However, unlike the worker threads that an `ExecutorService` uses, the threads of a `ForkJoinPool` use a work-stealing algorithm, which means that when a thread is free, it **STEALS** the pending work of other threads that are still busy doing other work.
- A `ForkJoinTask` object has three main methods, `fork()`, `join()`, and `compute()`. The **ORDER** in which you call the methods is **IMPORTANT**.

- You must first call `fork()` to queue the first subtask, then `compute()` on the second subtask to process it recursively, and then `join()` to get the result of the first subtask.

## Self Test

1. What of the following statements is true?
A. `RecursiveAction` is a subclass of `ForkJoinPool` .
B. When working with the Fork/Join framework, by default, one thread per CPU is created.
C. You need to shut down a `ForkJoinPool` explicitly.
D. `fork()` blocks the program until the result is ready.

2. Which of the following is the right order to call the methods of a `ForkJoinTask` ?
A. `compute()` , `fork()` , `join()`
B. `fork()` , `compute()` , `join()`
C. `join()` , `fork( )`, `compute()`
D. `fork()` , `join()` , `compute()`

3. When using a `RecursiveTask` , which of the following statements is true?
A. You can use the `invokeAll()` method instead of the `fork()` / `join()` / `compute()` methods.
B. You can use `ExecutorService` directly with this class.
C. An action is triggered when the task is completed.
D. `ForkJoinTask.invoke()` returns the same type as the generic type of `RecursiveTask` .

4. Given:

```
public class Question_28_4 extends RecursiveTask<Integer> {
   private int n;

   Question_28_4(int n) {
      this.n = n;
   }

   public Integer compute() {
      if (n <= 1) {
         return n;
      }
      Question_28_4 t1 = new Question_28_4(n - 1);
      Question_28_4 t2 = new Question_28_4(n - 2);
      t1.fork();
      return t2.compute() + t1.join();
   }
}
```

What is not right about this implementation of the Fork/Join framework?
A. Everything is right, it's a perfect implementation of the Fork/Join framework.
B. The order of the `fork()` , `join()` , `compute()` methods is not right.
C. This implementation is very inefficient, the subtasks will be very small.
D. It doesn't compile.

Open answers page

---

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

27. Concurrency 29. JDBC API